Programmazione asincrona & `fetch` API

Niccolò Maltoni
niccolo.maltoni@diennea.com

Programmazione asincrona

In JavaScript, poiché il motore è **single-threaded**, un compito pesante occupa interamente il thread principale, impedendo al browser di gestire eventi dell'utente (come i click) o di aggiornare l'interfaccia finché il compito non è terminato.

```
console.log('1. Richiesta dati avviata...');

const xhr = new XMLHttpRequest();
// Il terzo parametro 'false' rende la chiamata SINCRONA
xhr.open('GET', 'https://httpbin.org/delay/5', false);

xhr.send(null); // Il thread si FERMA qui per 5 secondi!
if (xhr.status === 200) {
  console.log('2. Dati ricevuti:', xhr.responseText);
}

console.log('3. Ora il codice può finalmente proseguire');
```

```
console.log('1. Richiesta dati avviata...');

const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://httpbin.org/delay/5', true);  // ASINCRONA

xhr.onreadystatechange = function() {
  if (xhr.readyState === 4) {  // Completata
    if (xhr.status === 200) {
      console.log('2. Dati ricevuti:', xhr.responseText);
    }
  }
};
xhr.send(null);  // Non blocca!
console.log('3. Codice prosegue immediatamente!');

console.log('4. UI reattiva subito!');
```

È chiaro che questo comportamento è indesiderabile.

La soluzione è delegare l'I/O al browser in background tramite un'operazione **asincrona**.

Callback: da sincroni ad asincroni

Come abbiamo visto con `XMLHttpRequest`, le operazioni asincrone richiedono di passare una **callback** che viene eseguita quando l'operazione è completata.

Ad esempio, proviamo a caricare uno script dinamicamente all'interno di una pagina:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('/my/script.js');
```

⚠ In questo modo, stiamo iniettando lo script, ma non abbiamo modo di sapere quando è stato caricato e se è stato caricato correttamente, dunque rischiamo di eseguire codice che dipende dallo script prima che sia pronto, causando errori.

Callback: da sincroni ad asincroni

Come abbiamo visto con `XMLHttpRequest`, le operazioni asincrone richiedono di passare una **callback** che viene eseguita quando l'operazione è completata.

Ad esempio, proviamo a caricare uno script dinamicamente all'interno di una pagina:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('/my/script.js', () => console.log('Script caricato!'));
```

Utilizzando la proprietà `onload` dell'elemento `<script>`, possiamo eseguire una callback quando lo script è stato caricato correttamente.

Però se volessimo caricare più script in sequenza?

Callback: da sincroni ad asincroni

Come abbiamo visto con `XMLHttpRequest`, le operazioni asincrone richiedono di passare una **callback** che viene eseguita quando l'operazione è completata.

Ad esempio, proviamo a caricare uno script dinamicamente all'interno di una pagina:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('/my/script.js', () => {
    console.log('Script caricato!');
    loadScript('/my/other-script.js', () => {
        console.log('Secondo script caricato!');
        loadScript('/my/third-script.js', () => {
          console.log('Terzo script caricato!');
        });
    });
});
```

Utilizzando la proprietà `onload` dell'elemento `<script>`, possiamo eseguire una callback quando lo script è stato caricato correttamente.

Invocando `loadScript` all'interno della callback del precedente, possiamo caricare più script in sequenza, ma questo porta a un problema di composizione…

Callback hell

Caricare più script in sequenza con callback, soprattutto in presenza di logica strutturata, diventa rapidamente illeggibile:

```
loadScript('/script1.js', (err1, script1) => {
  if (err1) { handleError(err1); }
  else {
    loadScript('/script2.js', (err2, script2) => {
      if (err2) { handleError(err2); }
      else {
        loadScript('/script3.js', (err3, script3) => {
          if (err3) { handleError(err3); }
          else {
            // Finalmente posso usare tutti gli script!
          }
        });
      }
    });
  }
});
```

Questo stile di codice è spesso chiamato **callback hell** o **pyramid of doom** a causa della struttura a piramide che si forma con i callback annidati.

Promise

Un approccio più moderno per gestire l'asincronia è utilizzare il pattern **promise**.

Una `Promise` è un oggetto JavaScript usato per gestire operazioni asincrone.

Esso rappresenta il **risultato futuro** di un'operazione asincrona che non è ancora disponibile, ma che lo sarà in un momento successivo.

Può avere **tre stati mutuamente esclusivi**:

Stato	Quando	Proprietà
Pending	Operazione in corso	Non completata ancora
Fulfilled	Operazione riuscita	Ha un valore (result)
Rejected	Operazione fallita	Ha un errore (reason)

> ⚠ **Una `Promise` è immutabile!**
>
> Una volta risolta o rigettata, **rimane in quello stato per sempre**.

Creazione di una `Promise`

Le Promise vengono create con il costruttore `new Promise(executor)`:

```
const myPromise = new Promise((resolve, reject) => {
  // executor function - eseguita IMMEDIATAMENTE
  console.log('Executor in esecuzione');

  // Simulazione operazione asincrona
  setTimeout(() => {
    const success = true;
    if (success) {
      resolve('Operazione completata!');  // → Fulfilled
    } else {
      reject(new Error('Operazione fallita'));  // → Rejected
    }
  }, 1000);
});

console.log('Promise creata (pending)');
// Output:
// Executor in esecuzione
// Promise creata (pending)
// (dopo 1s) → resolve() viene chiamato
```

L'executor viene eseguito **immediatamente**, mentre `resolve`/`reject` vengono chiamati **dopo** (asincrono).

“*Promisification*” di funzioni callback-based

Possiamo **sempre** convertire una funzione callback-based in una che ritorna `Promise`:

```
// Versione callback (vecchia)
loadScript('/my/script.js', (err, script) => {                    ①
  if (err) console.error(err);
  else console.log('Caricato:', script.src);
});

// Versione Promise (moderna)
function loadScriptPromise(src) {                                 ②
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {                            ③
      if (err) reject(err);
      else resolve(script);
    });
  });
}
```

① Callback-based: error-first pattern

② Wrapper che ritorna Promise

③ Delega alla funzione originale, traduce callback in resolve/reject

Gestire il successo della promise

Il metodo `.then(callback)` viene utilizzato per registrare una callback che viene eseguita in caso di **fulfill**:

```
const promise = new Promise((resolve) => {
  setTimeout(() => resolve('Dati ricevuti!'), 1000);
});

promise.then((result) => {
  console.log('Successo:', result);  // Eseguito quando risolto
});

console.log('Promessa in attesa...');

// Output:
// Promessa in attesa...
// (dopo 1s) Successo: Dati ricevuti!
```

`.then()` ritorna una nuova `Promise`, permettendo il chaining:

```
promise
  .then(r => console.log('First:', r))
  .then(() => console.log('Second'))
  .then(() => console.log('Third'));

// Eseguiti in sequenza quando la prima si risolve
```

Gestire il fallimento della promise

Il metodo `.catch(callback)` viene utilizzato per registrare una callback che viene eseguita in caso di **reject**:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('Errore di rete!')), 1000);
});

promise
  .then(result => console.log('Dati:', result))
  .catch((error) => {
    console.error('Errore catturato:', error.message);
  });

// Output (dopo 1s):
// Errore catturato: Errore di rete!
```

anche `.catch()` ritorna una nuova `Promise`, permettendo il chaining:

```
promise
  .then(r => console.log('First:', r))
  .catch(err => {
    console.error('Errore:', err);
    return 'Valore di fallback';  // Ritorna un valore per continuare
  })
  .then(r => console.log('Second:', r));  // Eseguito anche dopo l'errore
```

Gestire entrambi i casi

Il metodo `.finally()` registra una callback che viene eseguita **indipendentemente** da successo o errore:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('OK'), 1000);
});

promise
  .then(result => console.log('Successo:', result))
  .catch(error => console.error('Errore:', error))
  .finally(() => {
    console.log('Operazione terminata');  // Eseguito SEMPRE
  });

// Output (dopo 1s):
// Successo: OK
// Operazione terminata
```

Il caso d'uso più frequente è il **cleanup**: chiudere connessioni, fermare spinner di caricamento, etc.

Esecuzione parallela

In alcuni casi vogliamo eseguire più operazioni asincrone in parallelo con diversi comportamenti:

`Promise.all(...)` attende che **tutte** le promise si risolvano:

```
const p1 = fetch('/api/todos/1');
const p2 = fetch('/api/todos/2');

Promise.all([p1, p2])
  .then(responses => {
    responses.forEach((r, i) => console.log(`${i+1}: ${r.status}`));
  })
  .catch(err => console.error('Almeno una fallita:', err));
```

`Promise.race(...)` restituisce la **prima** completata:

```
const p1 = new Promise(r => setTimeout(() => r('Lento'), 3000));
const p2 = new Promise(r => setTimeout(() => r('Veloce'), 1000));

Promise.race([p1, p2])
  .then(result => console.log('Primo:', result));
// Output: Primo: Veloce
```

Un caso d'uso di `Promise.race` è implementare **timeout** per operazioni asincrone.

I limiti del pattern Promise

Le **promise** hanno migliorato il callback hell, ma con molte operazioni asincrone, anche le catene di `.then()` diventano verbose e difficili da seguire.

Inoltre, presentano alcuni **comportamenti non intuitivi** che possono causare bug.

Vediamone alcuni…

Passaggio di parametri

Cosa c'è che non va questo codice?

Osserva questo codice:

```
function delay(ms) {
  console.log(`Creando promise per ${ms}ms`);
  return new Promise(resolve =>
    setTimeout(() => resolve(ms), ms)
  );
}

// Vogliamo eseguire delay(1000) poi delay(2000)
delay(1000).then(delay(2000));
```

Qual è il problema?

Output:

```
Creando promise per 1000ms  ← Prima delay
Creando promise per 2000ms  ← Seconda delay parte SUBITO!
```

Cosa notiamo?

- Tutte le promise vengono create **immediatamente** (prima di "Fine")
- Non aspettano che la precedente finisca: eseguono in **parallelo**

Questa caratteristica è detta **eagerness** (*immediatezza*): le promise partono appena create!

L'uso corretto sarebbe:

```
delay(1000).then(() => delay(2000));
```

Di fatto, stiamo utilizzando una callback per ritardare la creazione della seconda promise.

Esecuzione sequenziale nei loop

Cosa c'è che non va questo codice?

Osserva questo codice:

```
for (let i = 0; i < 3; i++) {
  delay(1000).then(() =>
    console.log(`Step ${i}`)
  );
}
```

Output:

```
Creando promise per 1000ms
Creando promise per 1000ms
Creando promise per 1000ms
Step 0
Step 1
Step 2                    ← Tutte insieme dopo 1s!
```

Qual è il problema?

Il problema è sempre quello: tutte le promise partono **subito** al primo ciclo ed eseguono in **parallelo**!

Per eseguirle sequenzialmente dovremmo invece concatenarle con `.then()`:

```
let promise = Promise.resolve();
for (let i = 0; i < 3; i++) {
  promise = promise.then(() =>
    delay(1000).then(() =>
      console.log(`Step ${i}`)
    )
  );
}
```

Output:

```
Creando promise per 1000ms
Creando promise per 1000ms
Creando promise per 1000ms
Step 0                    ← Dopo 1s
Step 1                    ← Dopo altri 1s
Step 2                    ← Dopo altri 1s
```

Diventa molto più verboso e difficile da leggere!

Pattern async/await

Trattandosi di un problema comune, molti linguaggi di programmazione hanno introdotto costrutti per semplificare la scrittura di codice asincrono.

A partire da **ES2017**, JavaScript introduce le keyword `async` e `await` per risolvere questi problemi:

- **Leggibilità**: sintassi lineare che sembra codice sincrono, più facile da leggere e mantenere
- **Controllo dell'esecuzione**: `await` pausa l'esecuzione finché la promise non si risolve
- **Loop sequenziali**: `for`/`while` + `await` funzionano come ci aspettiamo
- **Parametri naturali**: non serve wrappare in arrow function

Vediamo il confronto diretto…

Confronto diretto: Promise vs Async/Await

Promise

```
// Sequenza
delay(1000)
  .then(() => delay(2000))
  .then(() => console.log('Fine'));

// Loop parallelo
for (let i = 0; i < 3; i++) {
  delay(1000).then(() =>
    console.log(`Step ${i}`)
  );
}

// Loop sequenziale
let promise = Promise.resolve();
for (let i = 0; i < 3; i++) {
  promise = promise.then(() =>
    delay(1000).then(() =>
      console.log(`Step ${i}`)
    )
  );
}
```

Async/Await

```
// Sequenza
await delay(1000);
await delay(2000);
console.log('Fine');

// Loop parallelo
for (let i = 0; i < 3; i++) {
  delay(1000).then(() =>
    console.log(`Step ${i}`)
  );
}

// Loop sequenziale (naturale!)
for (let i = 0; i < 3; i++) {
  await delay(1000);
  console.log(`Step ${i}`);
}
```

Funzioni `async`

La keyword **`async`** può essere anteposta alla dichiarazione di una funzione.

Una funzione `async` esegue automaticamente il *wrap* del suo valore di ritorno in una `Promise` risolta:

```
async function getValue() {
  return 1;  // Equivale a: return Promise.resolve(1)
}

getValue().then(result => console.log(result));  // 1
```

Similmente, qualsiasi errore lanciato viene trasformato in una `Promise` rigettata:

```
async function getError() {
  throw new Error('Ops!');  // Equivale a: return Promise.reject(new Error('Ops!'))
}

getError().catch(err => console.error(err.message));  // Ops!
```

In alternativa, possiamo anche ritornare esplicitamente una `Promise`:

```
async function getPromise() {
  return Promise.resolve(42);
}

getPromise().then(result => console.log(result));  // 42
```

Sospensione non bloccante con `await`

La keyword **`await`** può essere anteposta a qualsiasi espressione che ritorna una `Promise` per sospendere l'esecuzione in attesa di un risultato.

In particolare, essa permette di restituire il controllo all'event loop finché la `Promise` non è risolta o rigettata, **senza bloccare il thread principale**.

```
async function esempio() {
  console.log('Prima');

  await delay(2000);  // Pausa qui per 2 secondi

  console.log('Dopo 2s');
}

esempio();
console.log('Codice esterno continua!');

// Output:
// Prima
// Codice esterno continua!  ← Eseguito subito!
// (dopo 2s) Dopo 2s
```

`await` può essere usata solo all'interno di funzioni `async` (o anche top-level in browser moderni).

Unwrap automatico delle `Promise`

Come `async` esegue il *wrap* automatico del risultato in una `Promise`, `await` invece esegue l'*unwrap* automatico, restituendo direttamente il valore risolto:

- Se la promise è **risolta** → `await` restituisce il valore
- Se la promise è **rigettata** → `await` lancia un'eccezione

```
async function fetchData(url) {
  try {
    const response = await fetch(url);
    // ↑ await estrae response dalla Promise

    const data = await response.json();
    // ↑ await estrae data dalla Promise

    return data;  // Ritorna il valore estratto

  } catch (error) {
    // ↑ Cattura eccezioni da promise rigettate
    console.error('Errore:', error.message);
  }
}
```

Il risultato è un codice più lineare e facile da leggere, senza callback annidati o catene di `.then()`; risulta molto più simile a codice sincrono, ma con tutti i vantaggi dell'asincronia.

Lanciare errori con `throw`

In JavaScript, quando qualcosa va storto, usiamo **`throw`** per lanciare un errore:

```
throw new TypeError('Eta deve essere un numero');
throw new RangeError('Eta deve essere tra 0 e 150');
throw new Error('Errore generico');
```

`throw` interrompe l'esecuzione e passa il controllo al blocco `catch` più vicino:

```
function validaEta(eta) {
  if (typeof eta !== 'number') {
    throw new TypeError('Eta deve essere un numero');  // ← Interrompe qui
  }
  if (eta < 0 || eta > 150) {
    throw new RangeError('Eta deve essere tra 0 e 150');
  }
  return eta;
}

validaEta('ventiquattro');  // TypeError lanciato
console.log('Questo non viene eseguito');
```

Senza gestione, il programma "muore" e l'errore compare in console.

Catturare errori con `try` e `catch`

Con `try/catch`, il programma **non crasha** e possiamo gestire l'errore:

```
try {
  const valore = validaEta(-5);  // ← Lancia RangeError
  console.log('Eta valida:', valore);  // Non eseguito
} catch (error) {
  // ← Catturiamo l'errore qui
  console.error(`${error.name}: ${error.message}`);
  // RangeError: Eta deve essere tra 0 e 150
}

console.log('Programma continua normalmente!');  // ← Eseguito!
```

L'oggetto `error` contiene proprietà utili:

```
error.name        // "TypeError", "RangeError", "Error", etc.
error.message     // Messaggio descrittivo dell'errore
error.stack       // Stack trace (per debugging)
```

Cleanup con `finally`

Il blocco `finally` viene eseguito **sempre**, indipendentemente da errore o successo:

```
try {
  const valore = validaEta(25);
  console.log('Eta valida:', valore);
} catch (error) {
  console.error('Errore:', error.message);
} finally {
  console.log('Validazione terminata');  // Sempre eseguito
}

// Output (successo):
// Eta valida: 25
// Validazione terminata

// Output (errore):
// Errore: Eta deve essere tra 0 e 150
// Validazione terminata
```

Caso d'uso: ripulire risorse (fermare spinner, chiudere connessioni, etc.)

Validazione ed errori custom

Combiniamo validazione, `throw` e `catch` per gestire dati incompleti:

```
function readUser(json) {
  let user = JSON.parse(json);  // Può lanciare SyntaxError

  if (!user.name) {
    throw new Error('Dati incompleti: manca name');  // ← Throw personalizzato
  }
  if (!user.age) {
    throw new Error('Dati incompleti: manca age');
  }

  return user;
}

try {
  const user = readUser('{ "age": 30 }');  // JSON valido ma incompleto
} catch (error) {
  console.error(error.message);  // Dati incompleti: manca name
}
```

Gestione errori con `async/await`

Con `async/await`, usiamo lo stesso `try/catch/finally` per gestire errori:

```
async function fetchUserSafe(userId) {
  try {
    const response = await fetch(`/api/users/${userId}`);

    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);  // ← throw come in codice sincrono
    }

    const user = await response.json();
    return user;
  } catch (error) {
    // ← Cattura sia errori lanciati che promise rigettate
    console.error('Errore:', error.message);
    return null;
  } finally {
    console.log('Fetch completato');  // Sempre eseguito
  }
}

fetchUserSafe(999);
// Output:
// Errore: HTTP 404
// Fetch completato
```

I pattern sono identici tra codice sincrono e asincrono!

Fetch vs XHR: differenze

Abbiamo già visto `XMLHttpRequest`. `fetch` è l'alternativa moderna:

Aspetto	XHR	Fetch
Base	Callback/onload	Promise-based
Sintassi	Verbosa	Compatta
Errori	onload/onerror	.then()/.catch()
Body	responseText	response.json()

In pratica:

```
// XHR (vecchio)
const xhr = new XMLHttpRequest();
xhr.open('GET', url);
xhr.onload = () => {
  if (xhr.status === 200) console.log(JSON.parse(xhr.responseText));
};
xhr.send();

// Fetch (moderno)
fetch(url).then(r => r.json()).then(d => console.log(d));
```

Fetch GET: richiesta semplice

```
// GET di default
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(todo => console.log('Todo:', todo.title))
  .catch(error => console.error('Errore:', error));
```

Con async/await:

```
async function getTodo(id) {
  try {
    const response = await fetch(`/api/todos/${id}`);
    const todo = await response.json();
    console.log('Todo:', todo.title);
  } catch (error) {
    console.error('Errore:', error);
  }
}

getTodo(1);
```

`response.ok` e `response.status`

Importante: Fetch **non lancia errore** per codici di errore HTTP (e.g. 404, 500, etc.)

Dobbiamo controllare manualmente:

```
async function fetchSafe(url) {
  const response = await fetch(url);                                    ①

  console.log(response.status);    // 200, 404, 500, etc.
  console.log(response.ok);        // true se 200-299, false
  console.log(response.statusText); // 'OK', 'Not Found', etc.

  if (!response.ok) {                                                   ②
    throw new Error(`HTTP ${response.status}`);
  }

  return await response.json();                                         ③
}

fetchSafe('/api/todos/99999').catch(err => {
  console.error('Errore:', err.message); // HTTP 404
});
```

① Fetch non lancia errore per status 4xx/5xx

② Controllo manuale di `response.ok` (true se 200-299)

③ Solo se ok, parsing JSON

Best practice: controllare `response.ok` PRIMA di `.json()`

Pattern HttpError: errori HTTP custom

Creare una classe `HttpError` per distinguere errori HTTP da altri errori:

```
class HttpError extends Error {                                    ①
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

async function loadJson(url) {                                     ②
  const response = await fetch(url);
  if (!response.ok) {
    throw new HttpError(response);                                 ③
  }
  return await response.json();
}
```

① Classe custom che estende `Error`

② Helper che incapsula controllo `response.ok`

③ Lancia `HttpError` se status non ok

```
// Uso
loadJson('/api/user/invalid')
  .then(user => console.log(user.name))
  .catch(err => {
    if (err instanceof HttpError) {                                ④
      console.error('HTTP Error:', err.message);
    } else {
      console.error('Other Error:', err);
    }
  });
```

④ Possiamo distinguere errori HTTP da altri errori

Fetch POST: creare una risorsa

```
async function createTodo(title, userId) {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/todos', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify({
        title: title,
        completed: false,
        userId: userId
      })
    });
    if (!response.ok) { throw new Error(`HTTP ${response.status}`); }
    const newTodo = await response.json();
    console.log('Creato:', newTodo.id);
    return newTodo;
  } catch (error) {
    console.error('Errore creazione:', error);
  }
}

createTodo('Imparare async/await', 1);
```

Fetch PUT: aggiornare una risorsa

```
async function updateTodo(todoId, updates) {
  try {
    const response = await fetch(`/api/todos/${todoId}`, {
      method: 'PUT',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(updates)
    });

    if (!response.ok) throw new Error(`HTTP ${response.status}`);

    const updated = await response.json();
    console.log('Aggiornato:', updated.id);
    return updated;

  } catch (error) {
    console.error('Errore aggiornamento:', error);
  }
}

updateTodo(1, { completed: true, title: 'Task completato' });
```

Fetch DELETE: eliminare una risorsa

```
async function deleteTodo(todoId) {
  try {
    const response = await fetch(`/api/todos/${todoId}`, {
      method: 'DELETE'
    });

    if (!response.ok) throw new Error(`HTTP ${response.status}`);

    // DELETE spesso ritorna 204 (No Content)
    console.log('Eliminato:', todoId);
    return { success: true };

  } catch (error) {
    console.error('Errore eliminazione:', error);
  }
}

deleteTodo(1);
```